RDECOM TR 12-D-27

U.S. ARMY RESEARCH,
DEVELOPMENT &
ENGINEERING COMMAND

TITLE: Computational Fluid Dynamics Co-processing for Unsteady Visualization-Phase II SBIR Enhancement

AUTHOR(s): Andrew Bauer & Berk Geveci

COMPANY NAME: Kitware, Inc.

COMPANY ADDRESS: 28 Corporate Drive, #204
Clifton Park, NY 12065-8662

DATE (MONTH YEAR): September 2012

FINAL REPORT: Phase II SBIR Enhancement – Topic No. A07-010

Contract No. W911W6-08-C-0062

DISTRIBUTION STATEMENT A

Distribution Statement A: Distribution is approved for public release; distribution is unlimited.

Prepared for
U.S. ARMY RESEARCH, DEVELOPMENT & ENGINEERING COMMAND, AVIATION APPLIED TECHNOLOGY DIRECTORATE, FORT EUSTIS, VA 23604-5577

UNCLASSIFIED

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing this collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

1. REPORT DATE *(DD-MM-YYYY)* 30-09-2012	**2. REPORT TYPE** FINAL REPORT	**3. DATES COVERED** *(From - To)* May 2008 - September 2012
4. TITLE AND SUBTITLE Computational Fluid Dynamics Co-processing for Unsteady Visualization-Phase II SBIR Enhancement		**5a. CONTRACT NUMBER** W911W6-08-C-0062
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Berk Geveci, Andrew Bauer		**5d. PROJECT NUMBER**
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Kitware, Inc. 28 Corporate Drive, #204 Clifton Park, NY 12065-8662		**8. PERFORMING ORGANIZATION REPORT NUMBER** Topic #A07-010 Proposal #A2-3295
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) US Army Research, Development and Engineering Command (RDECOM) Aviation Applied Technology Directorate (AATD), Fort Eustis, VA 23604-5577		**10. SPONSOR/MONITOR'S ACRONYM(S)** AATD
		11. SPONSOR/MONITOR'S REPORT NUMBER(S) 12-D-27

12. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution Statement A: Distribution is approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

UL Note: No proprietary or limited information may be included in the abstract.

14. ABSTRACT

With the ever growing computational power available on the largest parallel machines, the ability to generate CFD simulation data is outpacing the ability to analyze and visualize it. This is due to the large disparity between the computational facilities (up to hundreds of thousands of cores) and the typical personal computer or visualization cluster. This project's goal was to begin the analysis and visualization of these large data sets on the parallel computers they are being run on. This reduces the file size and further computation required to fully analyze the results. In this report we detail the steps that were taken in order to enable ParaView and VTK to perform this coprocessing of simulation data and visualization and analysis information. The main highlights are GUI tools to easily specify what is to be outputted during the simulation run and a coprocessing library that gets connected to the simulation code that does the actual processing of the desired results. In addition, we demonstrate good parallel performance for the coprocessing library on an IBM BlueGene/L with up to 8,192 cores.

15. SUBJECT TERMS

Coprocessing, in-situ processing, CFD, visualization

16. SECURITY CLASSIFICATION OF: U			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Berk Geveci
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	21	**19b. TELEPHONE NUMBER** *(include area code)* (518) 371-3971

Phase II Technical Objectives

The major goal of the Phase II work was to develop a distributed coprocessing system capable of post-processing, at run-time, results of large, distributed simulations. The purpose of the system was to exploit the distributed processing capabilities of the Visualization Toolkit (VTK) and the ParaView framework in order to provide scalable post-processing capabilities. Furthermore, the system allows further analysis and interaction through the use of VTK, ParaView or any other VTK-based end-user application. Specific objectives include:

1. Develop an extensible and flexible coprocessing library
2. Develop CFD specific analysis and visualization algorithms
3. Develop tools for visualizing, analyzing and managing extracts
4. Integrate run-time visualization with coprocessing
5. Fully integrate and improve Helios coprocessing functionality
6. Improve particle tracking and other VTK components related to coprocessing
7. Minimize coprocessing library size
8. Next generation FieldView exporter
9. Scalability studies

In summary, the desire is to go from the classical way of post-processing as shown in Figure 1 to a two-step process of post-processing where much of the computational work is done during the simulation run as shown in Figure 2.

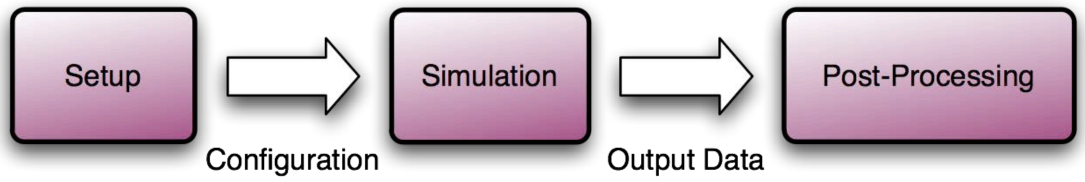


Figure 1: Classical simulation work flow.

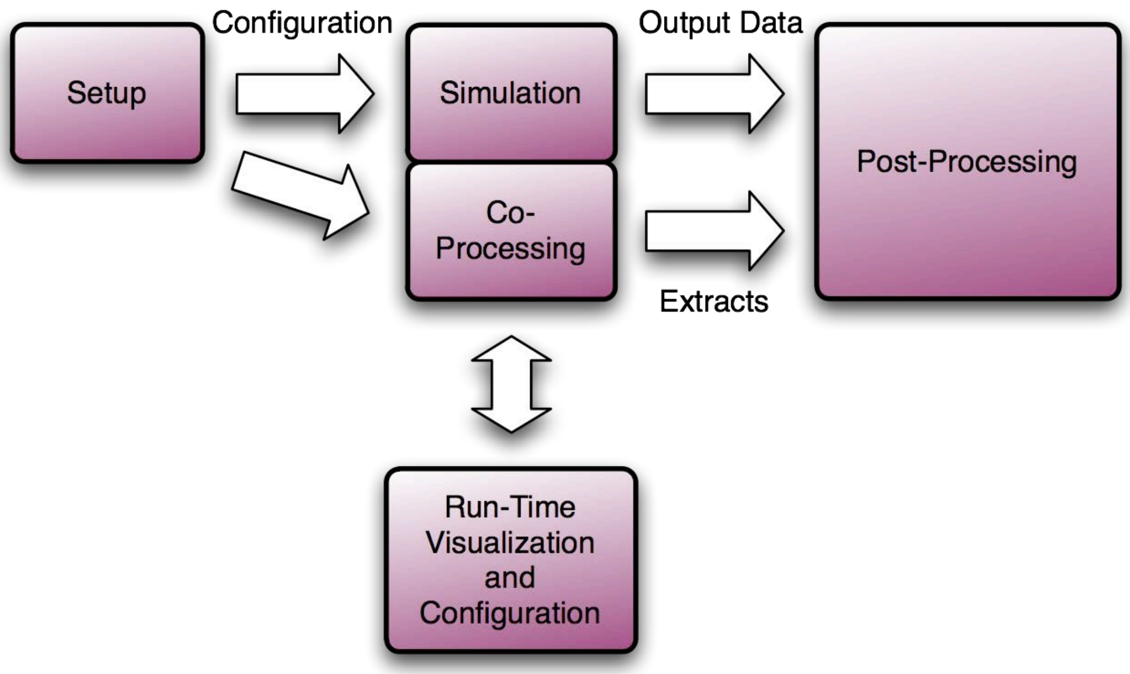


Figure 2: Work flow with coprocessing.

DISTRIBUTION STATEMENT A
Distribution Statement A: Distribution is approved for public release; distribution is unlimited.

Work Summary

The work that was done in order to achieve the project goal of performing coprocessing can be broken down into:

- VTK/ParaView developments
- ParaView GUI plugin to create coprocessing scripts
- ParaView coprocessing library
- Interfacing with simulation codes
- Performance/scaling tests

In addition to these goals we also wanted to promote coprocessing use during the project timeframe. In the following sections we detail the work to enable coprocessing through ParaView.

VTK Developments

Since this project was aimed at performing coprocessing of CFD simulations, the VTK improvements were aimed at algorithms and filters that are commonly used during analysis and visualization of fluid flow. This includes:

Octree-based locator: Previous point locators available in VTK suffered performance problems for highly graded meshes that are typically used for capturing the large gradients occurring in CFD simulations (e.g. boundary layer flow, shock, etc.) such as a grid around half of an airplane (shown in Figure 3). The basic problem was that they did not have a balanced distribution of points among the bins of the point locator (the uniform point locator) or the geometry of the bins had poor shape (the kd-tree point locator) as shown in Figure 4 for the grid in Figure 3. The solution to this was developing an octree-based point locator. The octree-based locator could adapt the bins locally to ensure that there was a balanced partitioning of the domain with respect to the points as well as locally subdividing any portion of the domain into 8 equally sized subsections. This resulted in having to examine less “nearby” points to determine the closest grid point/node to a given input coordinate value. Comparisons of the different point locators for 3 different grids are in Table 1 below.

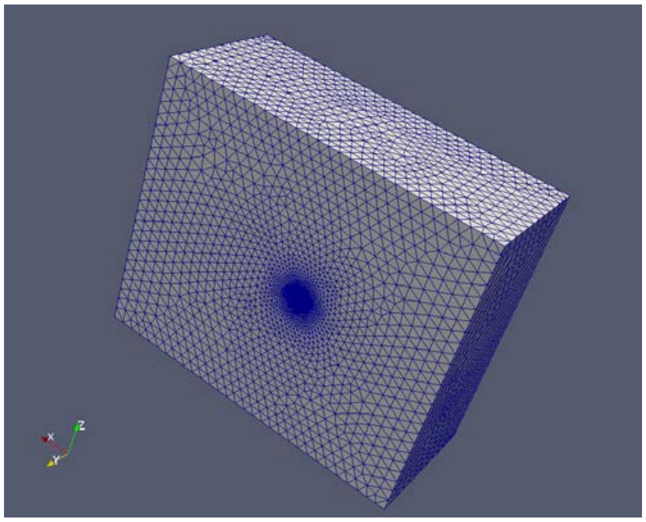


Figure 3: Grid surrounding an airplane.

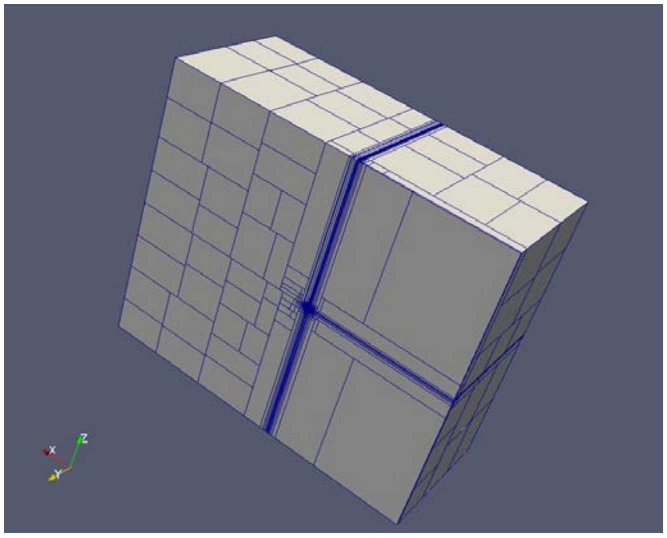


Figure 4: Kd-tree point locator bins

Grid	Uniform locator	Kd-tree locator	Octree locator
Figure 1 grid	7.78*10^-6	3.95*10^-6	1.42*10^-6
Figure 2 grid	5.39*10-6	4.18*10^-6	2.56*10^-6
Figure 3 grid	34.3*10^-6	8.7*10^-6	13.0*10^-6

Table 1: Locator search times in seconds

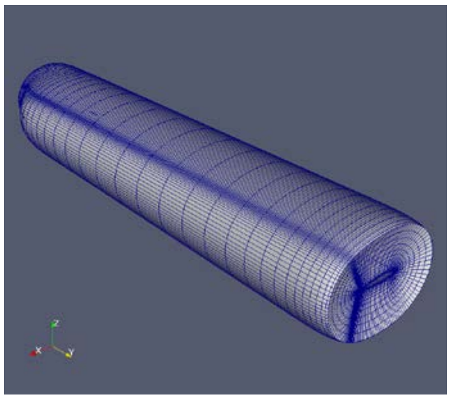


Figure 5: Grid surrounding a wing.

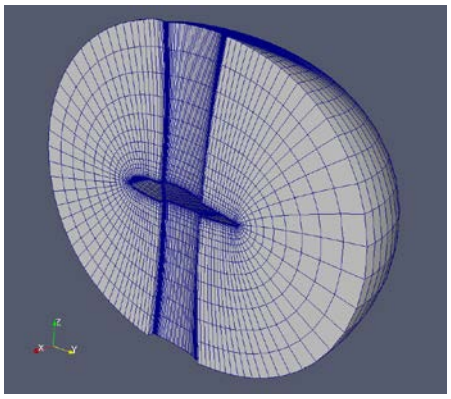


Figure 6: Grid around the end of a wing.

Incremental point locators: In the previous example, the locators were used for searching through grids with a static amount of points/nodes. Many filters wish to insert new points into the grid and need a point locator that can incrementally add new points to the existing ones. We label such locators as incremental since we require the locator to be able to handle both searching efficiently for existing points as well as being able to add new points without degrading the search performance drastically. The work that was performed was an implementation of an incremental octree-based point locator (IOPL). IOPL adopts an octree data structure to store point indices into a separate /external array of 3D coordinates. Specifically, it employs Incremental Octree Nodes (IONs) as the spatial subdivision primitives to build the search structure in a recursive top-down manner. The hierarchy takes the form of a tree-like representation by which a parent ION contains eight mutually exclusive (non-overlapping) child IONs. Each child is assigned with a rectangular volume (Spatial Bounding Box, SBB) and the eight children together cover exactly the same region as governed by their parent. In addition, the points falling within and registered, by means of point indices, in the parent ION are distributed to the child IONs for delegated maintenance. In fact, only leaf IONs, i.e., nodes without any descendants, actually store point indices while each ION, regardless of a leaf or non-leaf node, keeps a dynamically updated Data Bounding Box (DBB) of the inhabitant points, if any. Given a maximum number of points per leaf ION, an octree is initialized with an empty leaf ION that is then recursively sub-divided, but only on demand as points are incrementally inserted, to construct a populated tree. It is worth mentioning that inserting a point to a container leaf ION can incur recursive node sub-division since the points maintained by a

DISTRIBUTION STATEMENT A
Distribution Statement A: Distribution is approved for public release; distribution is unlimited.

leaf ION may fall within only one of the eight children in a heavily non-uniform fashion. As for implementation, the eight children of an ION are ordered by z-plane, then y-plane, and finally x-plane such that given a point the container child / octant can be rapidly determined via a comparison against the parent node center followed by three bit-shift operations. A lookup table is also exploited in combination with the parent node center to quickly set the SBB of each child. The ION class, i.e., vtkIncrementalOctreeNode, provides additional functionalities such as point-inside-node check, point-inside-node-DBB check, inter-node ancestor relationship check, recursive upstream propagation of updating the DBB and number of points for each node involved in the insertion of a new point, exporting the point indices in or under an ION, and computing the minimum squared distance from a point to an ION's DBB/SBB (optionally only inner boundaries for extended inter-node search). To find the closest in-octree point to a given point, IOPL begins with the root node and recursively navigates the octree to locate the leaf containing a reference point which is either the given point itself (if inside the octree) or an in-octree point roughly closest to the given point (if outside the octree). Then the points in the container leaf are checked against the given point by the squared distance to obtain the closest one. Based on this minimum squared distance, an extended within-radius inter-node search is performed, if necessary, in case of any non-empty neighboring node with a less distance to the given point. This extended within-radius inter-node search strategy also applies to the location of the closest point within a radius / sphere. To find the closest N points to a given point, the most\compact" (the lowest level) node containing roughly closest M (>= N) points is first determined by checking the point-to-node-DBB distance. These points are checked against the given point, inserted to, and sorted in a queue by the squared distance, followed by an extended within-radius inter-node search for any closer points. A relatively simple point location task supported by IOPL is to find all points within a radius through recursive access to octree nodes coupled with inter-point distance check. IOPL allows for point insertion in three increasingly complex modes, i.e., direct check-free insertion, zero tolerance insertion, and non-zero tolerance insertion. They all involve the location of the container leaf ION, whereas the latter two cases need to find the closest point, if any, within the leaf node. Furthermore, the third case requires an extended within-radius inter-node search, which, please note, differs from the one employed for locating the closest point(s) in that the radius is fixed (to the specified non-zero tolerance) in the former while it is dynamically updated / decreased in the latter. The IOPL class, i.e., vtkIncrementalOctreePointLocator, may either build an initial octree by processing a dataset or begins with an empty octree that is incrementally populated as points are inserted. It provides a polygonal representation of the varying octree boundary. In fact, we have developed five thorough tests with a challenging dataset (with 1620K\imbalanced" iso-value points) to verify its correctness and measure its performance. Initial timing results indicate that our incremental octree point locator is about four times faster than the current uniform bin-based point locator.

Improved AMR support in VTK: VTK's AMR support was improved in order to better support the in-memory AMR data structures of Samrai and Chombo. We spent time preparing a demonstration of coprocessing using the examples codes distributed with Chombo. We investigated the data transfer between the solver and coprocessing modules. Both Chombo and VTK need modifications to make the data exchange between the solver and co-processor possible. Chombo needs to provide method stubs in AMR.cpp (its time dependent solver driver) that periodically call the co-processor module and variables to hold the intervals. These modifications have been made, and will need to be permanently made part of Chombo if we are to make this demonstration available to the Chombo community. Going forward we should elicit the support of the Chombo developers in order to make this happen. We have begun to import the Chombo data using only the functionality currently available in VTK. To that end, the class vtkAMRBox was updated to be compatible with Chombo boxes. The class vtkUniformGrid has been equipped with Initializers that take an instance of vtkAMRBox and generate the ghost arrays that will needed for importing Chombo FArrayBoxes. With these modification we are able to successfully construct vtkHierarchicalBoxDataSets from collections of Chombo's FArrayBoxes. Currently in VTK there is only elementary support for Chombo's AMR data. Chombo's datasets differ from other AMR datasets in that levels are overset grids and block size can vary from block to block. Some of the important functionality, such as ghost cell and overset cell blanking support, has been provided in VTK but not implemented fully throughout the pipeline.

Improvements to VTK filters: With the coprocessing library built upon VTK and ParaView, it is crucial that the filters that are commonly used for CFD coprocessing work robustly and efficiently. Work continues on making the streamline integration more robust in parallel. In addition, we are in the process of designing an efficient task scheduling mechanism to optimize load balancing across processes. Other work includes improvement to the table-based clipper and surface LIC filters described in more detail below.

DISTRIBUTION STATEMENT A
Distribution Statement A: Distribution is approved for public release; distribution is unlimited.

- **Table-Based Clipper:** We developed an efficient Table-Based Clipper (vtkTableBasedClipDataSet) as an improved alternative to the existing filter (vtkClipDataSet). This new method takes advantage of a series of tetrahedralization only-on-demand lookup tables (LUTs) and cell-clipping LUTs to address various primitive cells, retaining the original grid topology whenever possible. A hash-based scheme is adopted to achieve rapid detection and removal of duplicate points. The table-based clipper outperforms its counterpart in both computational cost (one to two orders of magnitude of speedup) and memory footprint (consumption reduced to 14% ~20%). The new clipper is able to handling image data, polygonal data, rectilinear grids, structured grids, and unstructured grids.
- **Surface LIC:** We built on an initial implementation of a GPU-based SurfaceLIC with some enhancements and integrated it with VTK / ParaView. The GLSL fragment shaders were optimized by removing redundant rendering passes leading to a straightforward map from the LIC process to the GPU rendering pipeline. Texture masking was adopted for compositing the LIC image with the underlying geometry to expose part of the latter wherever the vector field is not defined. This addition increased the effectiveness of SurfaceLIC with the support of focus plus context visualization. To refine the LIC technique in terms of image contrast, a Laplacian high-pass filter was developed in a new fragment shader that lies between two LIC processes to accentuate flow streaks (Enhanced LIC). An option was provided for an adaptive switch in ParaView between one-pass regular LIC (upon user interaction) and two-pass enhanced LIC (for high-quality image creation with no user interaction).
- **FieldView Writer:** A filter was created that could output polydata (a grid with 0, 1, and 2D cell types) from VTK in a FieldView file format. Examples of this type include streamlines (fvp extension) and contours (fvuns extension). The writer works in parallel in that it does a reduction of the global grids into a single grid on process 0 and then writes it to file. While this is not optimal it should work fairly well for relatively small amounts of processes due to the reduced memory/file size of storing polydatas. This writer is now part of the Helios source code so that it will be available to all Helios users. Additionally, there is a ParaView plugin which allows the writer to be used when creating coprocessing scripts from any ParaView 3.14.1 and later installed version. We had hoped to extend the writer to the next-generation FieldView format but Intelligent Light was not very cooperative and this was not done.
- **Particle Tracker and Streakline Filters:** The particle tracker and streakline filters were modified to scale better as well as to work with the coprocessing library. They have been tested for a single uniform grid in serial and operate as expected. This is shown in Figure 7 for a swirling flow with seven streaklines. There are still some robustness issues working with Helios though. The first issue is that these filters must be called every time step. This causes problems since the simulation time step is very small compared to the fluid velocity (a time step length of $7.4*10^{-7}$ and a velocity on the order of 0.4 for the sphere regression test). The other issue is that for the sphere regression test the streakline filter causes Helios to crash after the twelfth time step.

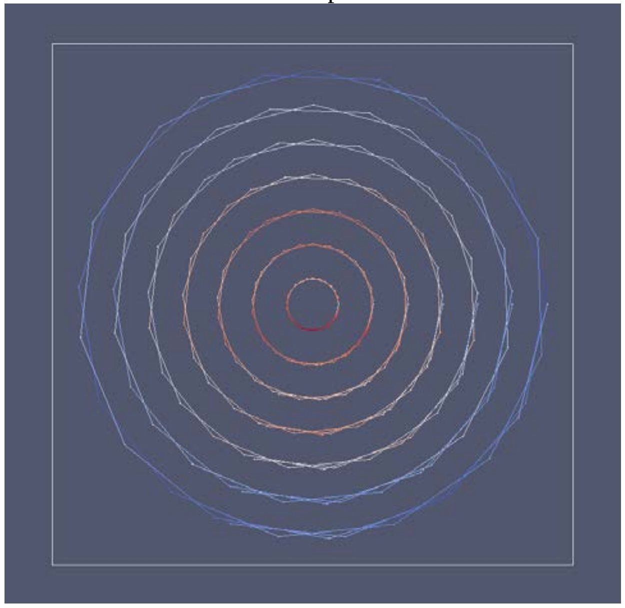


Figure 7: Strealines for a swirling flow.

DISTRIBUTION STATEMENT A
Distribution Statement A: Distribution is approved for public release; distribution is unlimited.

- **Gradient Filter and Q-Criterion Computation:** The vtkGradientFilter class was modified to support all grid types in VTK. This enabled it to be able to be used with multi-block datasets with all VTK grid types as well. This is needed for efficiently storing both the uniform and unstructured grids that are used in Helios. In addition, this class was modified to compute Q criterion.

ParaView Developments

In addition to the general improvements for VTK, there were also significant developments for ParaView. While these improvements were all aimed at better supporting coprocessing with ParaView, they also were general enough to be used elsewhere. Below we detail these improvements and indicate how they are specifically used for coprocessing.

Visualizing large datasets with streaming: The coprocessing stage typically generates large polygonal datasets which we would then like to visualize. Our goal is to provide a light-weight viewer that the user can start on his laptop or a local machine and simply open a file to start looking at the results. We can use ParaView for the visualization. However there are several challenges with this due to the nature of the data. Since the simulation is expected to run on servers with hundreds to thousands of nodes, we end up with a large extract that has hundreds to thousands of partitions. This greatly affects the interactivity as the complete dataset with all the partitions needs to be loaded before the rendering can commence. Ideally, we would like to load pieces of the dataset the user is looking at incrementally with increasing refinement and render them as soon as they become available. With this goal, we are developing a ParaView-based application with streaming support. The basic concept is simple: we have two threads, the master thread which is the standard ParaView client which does the rendering, GUI interactions, etc., while the second thread periodically loads pieces that are not yet available for rendering. As these pieces are loaded, the slave thread notifies the master thread which then incorporates the newly delivered piece into the rendering pipeline.

We use a special priority-assigner that can assign priority to pieces based on their refinement level and location in the user's view-frustum. Every time the camera changes, it re-evaluates the priorities and sets up a queue of pieces that are missing from the client and need to be fetched from the server. The slave thread periodically checks this queue and as and when pieces are placed on this queue, fetches them sequentially from the data-server. To further improve the interactivity, as new pieces are added to the rendering pipeline, the master thread simply creates new display lists for the new piece, leaving the display lists for other pieces intact.

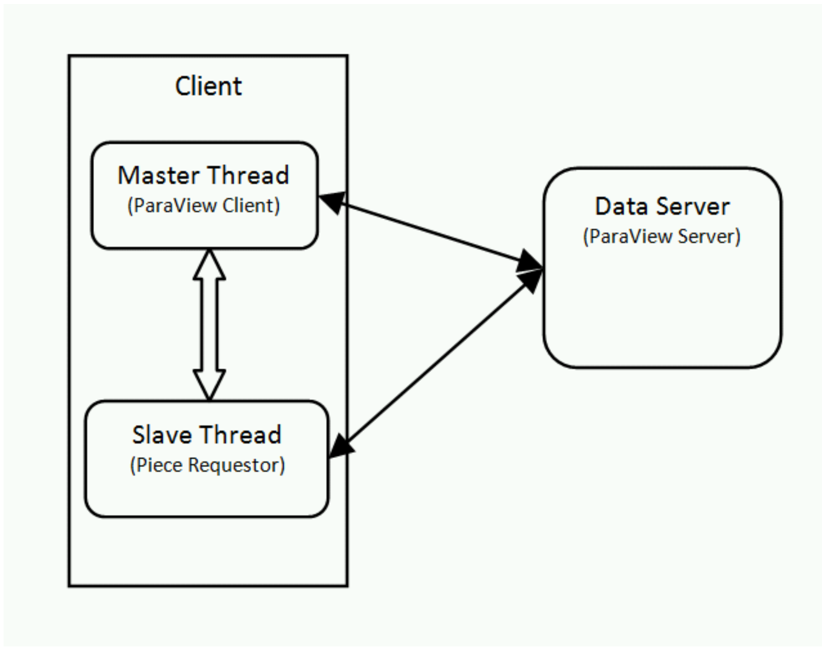


Figure 8: Streaming design.

The above figure shows the setup. The data-server is a regular ParaView server which provides the data. The client communicates with the root node of the server over two socket connections: one is the standard ParaView client-server communication channel over which ParaView client-server communication takes place; other is the channel over which the slave thread requests pieces from the server. The server responds to

communication over both channels responding to one at time, thus in effect serializing the requests. This works to our advantage since the slave thread as such cannot fetch pieces from the server side unless the pipeline has been updated which is done by the master thread.
Next we are working on fortifying the VTK pipeline so that it is possible for the data reader to read only the data associated with the pieces that are currently being requested by the client. This should further improve the client response time since the reader does not have to read all the data at the beginning of the visualization as is the case with conventional ParaView/VTK pipelines.

Streaming geometry viewer: While the coprocessing library can vastly reduce the amount of disk space used to store the desired results, there can still be large amounts of files as well as a computationally prohibitive amount of rendering to be done. The traditional ParaView based solution for this problem would be to move the files to a visualization cluster, run a pvserver process on it and then connect to it with a client running on the user's desktop. In this mode ParaView supports local as well as remote rendering. There are problems though with both of these rendering modes which include:

- Local Rendering:
  - Entire geometry must be fetched to the client before rendering
  - Entire geometry must fit on the client
  - Interactivivity suffers to due to large geometry
- Remote Rendering:
  - Rendering capabilities required on the server side, which limits how many server processes can be run on the cluster at the same time
  - For large geometries interactivity suffers
- Local/Remote Rendering:
  - The whole dataset needs to be read from disk before rendering can begin

The streaming geometry viewer was developed to address these issues. The highlights of the application are:

- User starts the pvserver on the visualization cluster and connects to it using the ParaView-based Geometry Viewer Application (Viewer)
- The Viewer supports local rendering alone. However it does not request the entire geometry for rendering. It uses a priority based ordering to requests pieces that are currently visible to the user
- The Viewer also supports LODs, so it can start with lowest resolution geometries, for better interactivity
- The Viewer is memory bound, it does not bring in more geometry when a predefined memory limit is reached. However using priorities, it tries to optimally use the available memory by discarded hidden pieces, focusing on the pieces in focus etc.

For implementation of the streaming geometry viewer, we assume that we will be dealing with polygonal datasets with lots of pieces since coprocessing is run in-line with same number of cores as the original simulation. During coprocessing we decided to use the large computational power to generate geometries with varying levels of details for each piece. All this will be packaged into a single mutliblock data set with the following structure, where LOD0 is the highest resolution geometry.

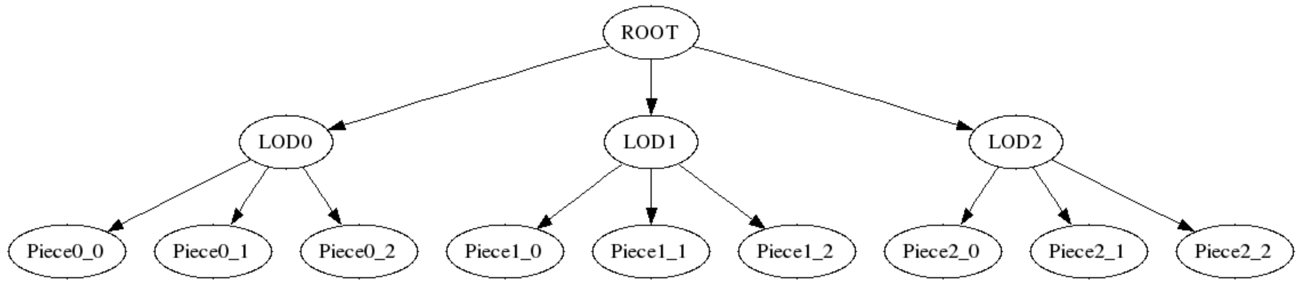


Figure 9: Streaming Geometry Viewer Structure.

One of the most important things we wanted to ensure was interactivity. For that we opted for a two thread approach. The master thread is the standard ParaView-based application which sets up the visualization/rendering pipeline, representations, views, etc. We added support to vtkCompositeDataPipeline for a data-source to announce the structure of the composite dataset it is going to produce. We also added capabilities to make a request for a particular block from the composite dataset. Thus, now, with a special reader that supports these new constructs, we can support reading of only the requested blocks. We created a special representation that does not request the entire composite dataset from the input. Instead, by default it simply requests the lowest LOD nodes. Thus when a dataset is opened up in this view, you merely see the

DISTRIBUTION STATEMENT A
Distribution Statement A: Distribution is approved for public release; distribution is unlimited.

lower LOD, which would ideally be just the outline for each piece or something very close to that. The second thread is responsible for requesting additional blocks from the server side, one at a time. This thread only works when the application is idle, so when the user is busy applying filters or interacting, we do not ruin the experience.
There is additional intelligence to improve rendering performance on the client side. We are creating separate display lists of each block. So as and when new blocks are delivered to the client, we do not need to create the display lists for the entire geometry. It is done for the newly delivered block instead. This is implemented as a custom ParaView application with a ParaView plugin that needs to be loaded on the server side (the client application links the plugin in, so it will always be loaded).

Python support: The first part of the Python work was creating a simplified Python scripting interface to ParaView. While ParaView always supported scripting, initially it was done with Tcl as the scripting language. As Python became more popular as a dynamic language for scientific computing, Python support was added to ParaView. Part of the work for this was separating the user interface from the core functionality as shown in Figure 10.

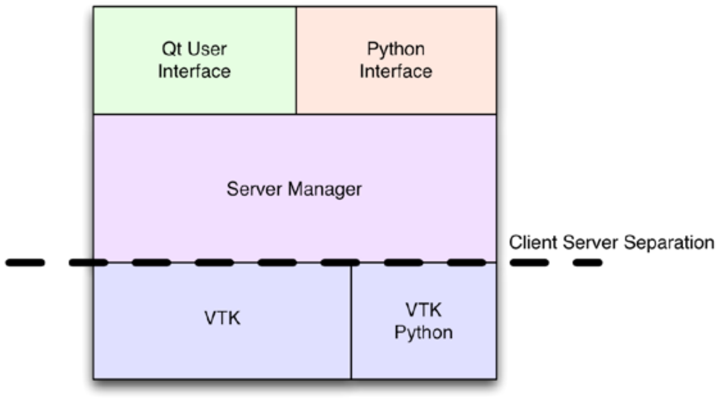


Figure 10: ParaView Architecture.
The advantage of this design is that all access to the ParaView core goes through the same programming interface. This means that the graphical user interface code and the Python layer both have access to the same functionality. However, the Server Manager was developed with developers in mind, not end users. This means that the Server Manager API has a steep learning curve and is not easily accessible to end users. In order to address this issue, we have been developing a thin Python layer over the Server Manager API to make it easier for users to write Python scripts that drive ParaView. The following is an example that demonstrates the Python interface.

```
from ParaView.simple import *
ExodusIIReader(FileName="can.ex2") # Create the reader for the Exodus file
Slice() # Create a slice filter
Show() # Turn the visibility of the slice filter on
Render() # Render the scene
```

This example demonstrates only a tiny subset of the Python API. The ParaView Python modules are self documenting through the Python documentation system. Also, we are in the process of updating the online documentation at http://ParaView.org/Wiki/ParaView/Python Scripting.
With the simplification of ParaView’s Python interface, we expect that some advanced users will be able to write their own Python scripts. There is still a bit of a learning curve though in order to do it if starting from scratch. The improvement that we worked on next was being able to save a trace of the current state of the ParaView GUI in a Python script. The idea was that the user could operate the GUI to obtain a rough outline of what they wanted to be scripted and then by tracing the user’s actions, output a script from ParaView that could easily be modified to achieve the desired scripting result. For example, when a new source or filter appears in the pipeline browser, the trace module observes ParaView’s proxy registration event and generates the appropriate Python code for constructing the same object. The generated trace code uses the Python API in the ParaView.simple module discussed above. The ParaView.simple module is a procedural styled Python API with commands meant to mimic the actions a user would perform in the GUI. For example, to make an object invisible in the GUI the user just clicks the visibility icon. In Python, a call to Hide() is sufficient. There are no required arguments to the Hide() function because the ParaView.simple module has a concept called the active

DISTRIBUTION STATEMENT A
Distribution Statement A: Distribution is approved for public release; distribution is unlimited.

source. Function calls in ParaView.simple operate on the active source when no specific object is specified. The active source is always (and automatically) set to the last source proxy used.
The code generated by the trace module uses the active source (and active view) concept developed for ParaView.simple. Whenever possible, the generated trace code calls functions defined in ParaView.simple without supplying source arguments ensuring that the active source is used. This conforms with the procedural style code and also makes the code more flexible. It is easier to recycle code when it describes actions without referring to the specific objects that the actions are performed on. The usage of active objects allows the module to be used to record macros. A user can build an initial pipeline, then start trace and perform a few actions and then save the trace output as a macro. Since the recorded trace uses active objects, the trace script does not have to specifically refer to any of the objects in the initial pipeline. This allows the trace code to be used as a macro on other pipelines in the future.
In normal execution mode the trace will record properties when they are modified from their default values. The trace can also run in verbose execution mode. In this execution mode all properties of a proxy are included in the trace. This may be useful to a user interested in exploring the full possibilities for customization provided by a given proxy. Verbose execution mode can also be used to initialize the current property values on a proxy that existed before trace was started by the user. Normally the property values would not be captured because they would not be modified after trace was activated.

Cross compiling ParaView with minimal builds: With the coprocessing library now being part of ParaView, in order to enable coprocessing we must cross compile the VTK and ParaView libraries for the operating systems running on the supercomputers' compute nodes. During the project we have streamlined the process of using CMake to cross compile VTK and ParaView for the IBM BlueGene/L operating system and the Cray Compute Node Linux operating system. While both systems support the GCC compiler toolchains, the IBM XLC compiler is preferred on BlueGene, and the PGI compiler is preferred for Cray CNL. With our effort we can now build ParaView using any of the three compiler toolchains.
The ParaView coprocessor contains a dynamic interpretive layer called client-server-stream. This interpretive layer allows the coprocessor to construct a data processing and visualization pipeline containing any filter in ParaView's arsenal. In practice however, a coprocessing routine is likely to use only a fraction of what is available. This means a standard build of the ParaView coprocessing library will contain, in binary code, a large number of unnecessary filters. If the coprocessing routine requires only data processing and no visualization then it is not necessary to include any rendering code at all. The goal of this work has been to build the coprocessing library so that it contains a minimal set of ParaView filters, therefore reducing the amount of memory required by the coprocessor when combined with a simulation code.

Reducing the binary executable footprint: The easiest way to reduce the size of the coprocessing binary code is to use static linking. The ParaView and coprocessing libraries are compiled as static archive libraries. When the simulation code links against the static libraries to create the final executable the linker ensures that only routines that are referenced are included. The problem is that the client-server-stream interpretive layer must reference all of the ParaView filters in case the coprocessor wants to instantiate one of them. The solution is to create a custom list of ParaView filters that should be included for a given coprocessor run, and limit the interpreter to these filters. The ParaView plugin generates a coprocessor script from a pipeline constructed with the ParaView client. We developed a parser tool that then generates a list of filters and rendering actors used by the pipeline. Since filters may depend on other filters, it is necessary for a secondary parsing tool to generate a final list that contains all filter dependencies. Dependencies are most common between rendering elements, so a routine that requires rendering usually results in a dependency list that is longer than pure data processing pipelines. The final size of an executable linked with the coprocessing library and its dependencies, compiled with the IBM XLC compiler with code optimization options enabled, is greater than 100 megabytes. After limiting the client-server-stream interpreter to only the ParaView filters required by a basic coprocessing routine (and the filter dependencies), the executable is reduced to 25 megabytes. While this technique successfully reduces the final executable size, it requires that a minimal filter set is generated for each coprocessor routine, and the coprocessor library is re-linked for each filter set.
In addition to the above work, modularization of both VTK and ParaView has been done for their next major releases, 6.0 and 4.0 respectively. This was done to reduce dependency and to enable turning off modules that aren't needed for certain use cases. For example, if outputted images are not needed for a coprocessing simulation output, all of the rendering libraries can be excluded from the ParaView build.

CoProcessing Library Redesign

DISTRIBUTION STATEMENT A
Distribution Statement A: Distribution is approved for public release; distribution is unlimited.

In order to move from the prototype that was developed in Phase I of the project to a more usable interface, we chose to redesign the coprocessing library. The new design was based on the assumption that a programmer that is familiar with the simulation source code would write the adapter to create VTK grids with the desired simulation fields defined over them. We feel that while VTK itself is a fairly large piece of software, the parts of it needed to construct grids and fields is quite small and straightforward for those people that have a reasonable amount of experience writing grid based simulation codes. This does assume that they are comfortable with C++ though. Once the adapter has been written, the end users are only required to create their coprocessing pipelines through the ParaView GUI plugin (discussed below) of which no programming is required. From this vision of the coprocessing library, the basic components of the design include:

1. Creating a Python script interface as well as ParaView type interface for creating pipelines for the coprocessing library to execute.
2. An advanced interface to describe what information is needed by the coprocessing pipelines each time step back to the adaptor.

Initially it is expected that most users that are trying to run their simulations with coprocessing will be more comfortable creating a coprocessing pipeline in ParaView. In order to do this, they will require an input file from one of their simulations to set up the pipeline. This pipeline setup process will allow the user to only create valid pipelines. In addition to the ParaView state to store the pipeline information, the plugin adds additional information that queries the user on how often each pipeline needs to be run as well the output format and naming conventions for the files.

While creating the coprocessing pipelines in ParaView should be fairly simple for the average user, it may not be flexible enough for advanced users. Simple examples of this type of use are contouring a scalar value based on the minimum and maximum values at that time step or outputting slice planes near where flow separation around a wing has occurred. If the user desires this type of advanced functionality where the coprocessing outputs are dependent on quantities computed from the simulation then the plugin generated Python script can be used as a starting point to create the coprocessing pipelines. After that, the user can edit the script to add in the additional logic to obtain the desired output. The advantage of this approach is that the user has much more control over what is outputted resulting in a combination of less overall file IO and more focused visualization results which will make it easier and more reliable to capture salient flow features.

While creating the proper visualization information is the most important aspect of the coprocessing library, the other main consideration is how efficiently the coprocessing operations are performed. This also includes the efficiency of the adapter. An important aspect in this efficiency is determining what information is necessary for a coprocessing pipeline to complete its tasks and passing this information back to the adapter. The reason for this is that while the simulation can provide the full grid and fields every time step, it may not be needed every time step and there is a computational cost to adapt/convert the simulation data structures into native VTK data structures. By passing back information from the coprocessing pipelines to the adapter it may be determined that no coprocessing is necessary during specific time steps and all data conversion can be skipped. A similar situation for making the adapter more efficient is when only a subset of the fields that are available through the adapter are needed by the pipeline. In this case a coprocessor can pass this information back to the adapter and the adapter can skip converting the field into VTK format. Of course as this is an advanced feature and it can be skipped for those that are just interested in testing out the coprocessing library without putting in a lot of effort in optimizing the adapter.

For the current design of the coprocessing library (not including the ParaView plugin for creating pipelines), the main classes are:

- **vtkCPDataDescription** A container class for storing the time, time step, grid, and field description (name,   number of components, type) for passing between the pipelines and the adapters. The time step and time are set by the adapter in order to pass it to a coprocessing pipeline to determine if coprocessing needs to be done that time step and/or time. If the pipeline determines that coprocessing is necessary, it passes back the field description for fields that are needed by the pipeline. If coprocessing is necessary then the adapter creates the grid with requested fields and passes this information back to the pipeline for execution. Note that the vtkCPDataDescription object is passed from the adapter to the pipelines through the vtkCPProcessor object instead of directly from the adapter to the pipelines themselves.
- **vtkCPPipeline** Abstract base class for pipelines with interface functions for querying whether the pipeline needs to be run, executing the pipeline, and finalizing the pipeline.
- **vtkCPPythonPipeline** Concrete subclass of vtkCPPipeline that is initialized with a Python script file that is used to specify when the pipeline is to be run as well as for executing the scripted pipeline.

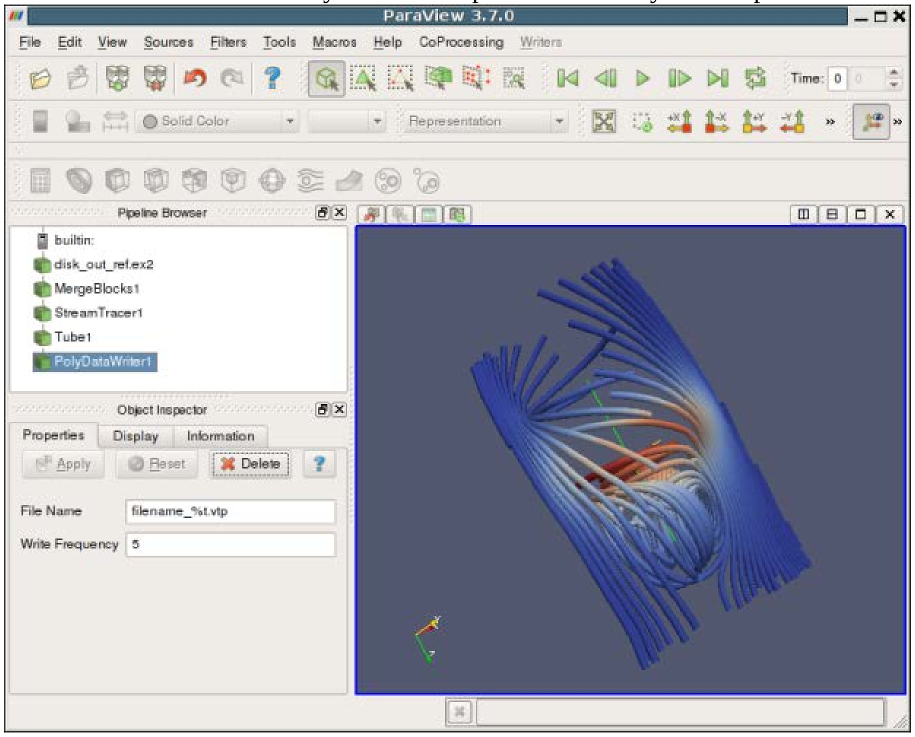
DISTRIBUTION STATEMENT A
Distribution Statement A: Distribution is approved for public release; distribution is unlimited.

- **vtkCPProcessor** The main class for storing pipelines. Multiple pipelines can be stored in this class so that they can be easily reused without having to create a single specific pipeline for each simulation run. The adapter only needs to directly interact with this class.

While the coprocessing library is naturally in a good state at the end of the project due to the effort spent, we can only expect it to flourish if users find the feature helpful as well as reliable. Later on we will cover how we are promoting ParaView's coprocessing features. For now though we have added some tests to ensure that changes to ParaView and/or VTK do not cause the coprocessing library to crash. As is normal for Kitware's open-source projects, the tests are run often (both nightly and after modifications to the repository) and are available for anyone with an internet connection to view. In addition to the testing, we have also created a test harness to easily diagnose problems with the coprocessing library. This is being used to demonstrate the library as well as giving a simple way to examine problems without having to compile the simulation code that is feeding input to the coprocessing library.

ParaView Plugin for Creating CoProcessing Scripts

While the coprocessing library can be a major tool in post-processing results from large simulation runs, without a simple way to specify the coprocessing extracts the library would be used very little, if at all. We are using the Python trace functionality discussed above to create coprocessing scripts. To keep this simple, we have developed a ParaView plugin that allows the user to setup the visualization pipeline using the ParaView GUI and then export it as a python script that is suitable for coprocessing. The user opens up a sample dataset in ParaView. This dataset simply needs to be of the same type as that produced by the simulation. This ensures that the pipeline that user will setup remains valid during the coprocessing run. The user then sets up the visualization pipeline using the ParaView GUI. If the simulation provides multiple data inputs, then the user should load multiple datasets and set up processing pipelines for each of them. If the user is interested in saving out data results from the visualization pipeline, then he can create a writer from the Writers menu as if it were a simple filter. The properties panel for the writer allows the user to choose a file name. One can use "%t" as a special character in the filename that will be replaced by the time step number at run-time. "Write Frequency" can be used to save results every i-th time step instead of every time step.



File Name filename_%t.vtp
Write Frequency 5

Figure 11: ParaView showing a sample visualization pipeline.

Note the CoProcessing and Writers menus added by the coprocessing plugin. Once the visualization is setup, the user can export the state as python using the Export option from the CoProcessing menu. This will pop up a wizard that will guide the user through the steps involved in exporting this state. In first step, after a welcome screen, the user chooses which data sources in the visualization pipeline that he has setup are to be replaced by the dataset provided by the simulation. As shown in Figure 12, the user simply selects the sources listed in the left list view and moves them to the right list view using the "Add" button. There must be at least one data source added to the right pane to proceed further in the wizard.

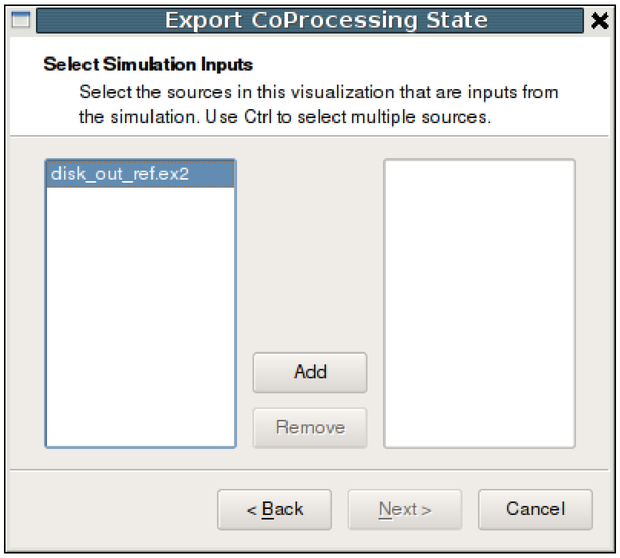
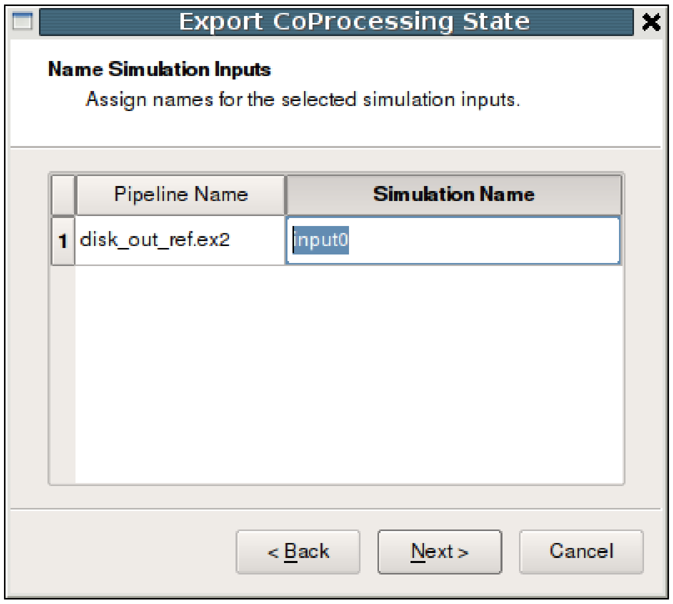

Export CoProcessing State
Select Simulation Inputs
Select the sources in this visualization that are inputs from the simulation. Use Ctrl to select multiple sources.
disk_out_ref.ex2
Add
Remove
< Back
Next >
Cancel

Figure 12: Specification of data sources.

Once the user has identified the data sources, on the next page he must indicate how they correspond to the inputs from the simulation. The adaptor names each of the inputs from the simulation which must match what is specified here.   This is shown in Figure 13.

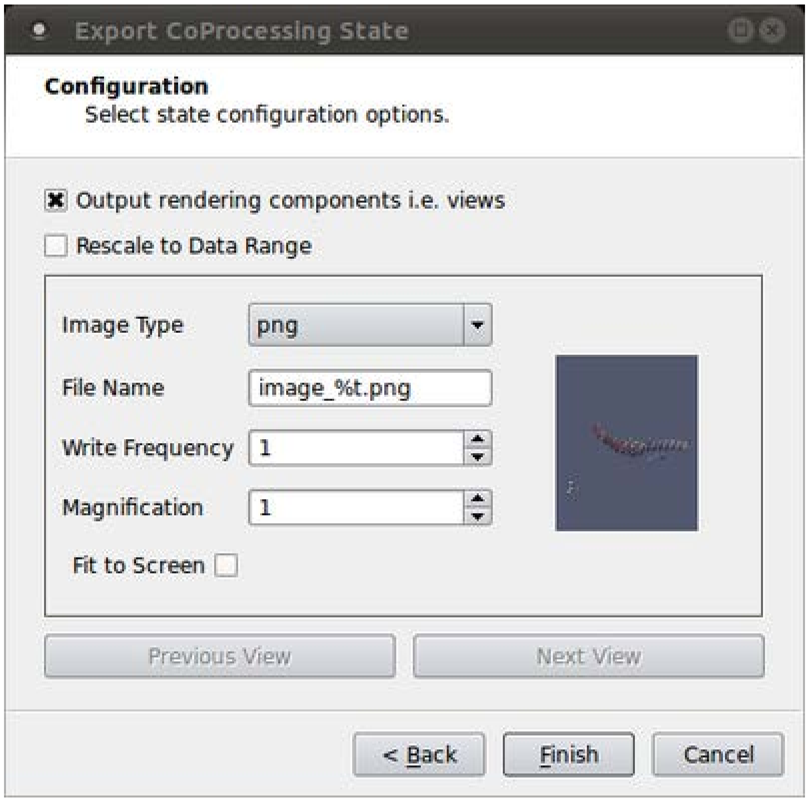

Export CoProcessing State
Name Simulation Inputs
Assign names for the selected simulation inputs.
< Back
Next >
Cancel

	Pipeline Name	Simulation Name
1	disk_out_ref.ex2	input0

Figure 13: Linking of source to coprocessing pipeline.

The final page allows the user to specify any images to be exported in addition the data that was specified earlier to be written out. Similar to the data writers, there is an option on how often each rendered images will be outputted. Additionally, for each rendered image the user can specify to the data objects to the viewing screen, the image type, and a magnification factor. An option for all outputted images is to rescale the pseudo-coloring to the current data range.

DISTRIBUTION STATEMENT A
Distribution Statement A: Distribution is approved for public release; distribution is unlimited.



Configuration
Select state configuration options.

[✓] Output rendering components i.e. views
[ ] Rescale to Data Range

Image Type: png
File Name: image_%t.png
Write Frequency: 1
Magnification: 1
Fit to Screen [ ]

Figure 14: Options for exporting the Python script.

The exported python script includes a global function with signature coprocessing (input, timestep) that takes two arguments, the first one is a map with named inputs from the simulation, while the second argument is the timestep number. The coprocessing library uses this function as the entry point for the visualization/coprocessing script. The crux of this function involves commands to set up the visualization pipeline as done in the ParaView GUI. If there were any writers in the pipeline, then this function ensures that the results are saved out respecting the frequency chosen by the user in the GUI. The following is a sample Python script file saved by this wizard.

```
try:
   paraview.simple
except:
   from paraview.simple import *

cp_writers = []
def cp_create_writer(proxy_ctor, filename, freq):
   # internal function to create writer proxies
   return writer

def co_processing(inputs, timestep):
   # Entry point for the coprocessing.
   global cp_writers
   cp_writers = []
   disk_out_ref_ex2 = inputs.__getitem__( "input0" )
   MergeBlocks1 = MergeBlocks( guiName="MergeBlocks1" )
   StreamTracer1 = StreamTracer( Ã– )
   Tube1 = Tube(Ã– )

   PolyDataWriter1 = cp_create_writer( XMLPPolyDataWriter, "filename_%t.vtp", 5 )
   # additional initialization of state

   # write results periodically
```

```
for writer in cp_writers:
    if timestep % writer.cpFrequency == 0:
        writer.FileName = writer.cpFileName.replace("%t", str(timestep))
    writer.UpdatePipeline()
```

The Python script has code to setup the visualization pipelines exactly as they were in the GUI, including the parameter values for different filters. Since the data set generated by the simulation may be different than the toy dataset used during the staging, one may want to update the script so that data range dependent filter parameters are updated correctly at run-time. This plugin makes it easier for users to quickly setup visualizations using ParaView and generate Python scripts suitable for coprocessing for further tweaking.

ParaView Run-Time Visualization

While the images and data files will be written out during coprocessing, it still can be cumbersome for the user to examine those results in real time. To help the user in this regard, we have developed a run-time visualization system for the coprocessor. The run-time visualization system allows a user's computer running the ParaView client to observe coprocessor results as they are generated by the executing simulation code. The data extracts produced at the coprocessor nodes are first reduced to a smaller subset of the coprocessor nodes and then sent via TCP connections to a running ParaView server. The system also supports data transfer via hard disk. Extracts may be written to a shared file system by the coprocessor where they can be accessed by a reader on the ParaView server. The coprocessor uses an M to N reduction (M>N) algorithm that minimizes wait time for point-to-point transfers. The M number of compute nodes are broken into groups where each group sends the data extract to their group's root node. The groups' root nodes are then partitioned into further groups and sends continue until the extracts have been relocated to N number of nodes. If the coprocessor is connected to a single process ParaView server then N is allowed to be 1. The reduction algorithm can be used prior to disk I/O as well. Instead of performing disk I/O on M coprocessor nodes we can move the data to N nodes to perform I/O. This has benefits on systems like BlueGene/L where I/O operations are performed on I/O nodes that are shared among sets of compute nodes. It is possible to choose the N compute nodes that collect all the data in a way to optimize the I/O speeds with respect to the I/O node organization. As the simulation code executes, a ParaView client connects to the same ParaView server that receives data extracts from the coprocessor. The ParaView client is able to visualize the extracts as they arrive from the coprocessor. While the run-time visualization helps in giving immediate feedback to the user, often as a result of this feedback the user will want to modify parameters in the Python script. Because of this, we also allow the user of the ParaView client to adjust parameters of the filters that compose the coprocessor pipeline. The adjusted parameter values are routed through the ParaView server to the coprocessor which will then use the updated values for the next cycle. A video of this functionality is available at http://paraview.org/Wiki/CoProcessing#Live_visualization.

Interfacing with Simulation Codes

For this project we focused our efforts on interfacing coprocessing with Helios. The coprocessing tools have been integrated into Helios's main code branch and have been going through testing by the main Helios developers. This has helped harden the system. We have also integrated a bunch of 'canned' coprocessing scripts inside of Helios. These are essentially preconstructed pipelines that only require setting a couple of parameters to get desired output. The current types of canned scripts are slices, contours/isosurfaces, streamlines, streaklines, particle paths, internal boundary extraction, and taps (i.e. sensors embedded in the domain to give result information). In addition to these canned scripts, the user can specify that vorticity and Q criterion should be included in addition to the simulation's independent variables, pressure, energy and momentum. Parameters for these canned scripts include output frequency and format as well as script specific information like seeds for the flow lines, isosurface values for contours, etc. The idea of the canned scripts is for Helios users to be able to get familiar using the coprocessing tools without a deep learning curve.
Besides the code integration, Helios was built and tested with the coprocessing tools on a variety of systems to ensure cross-platform functionality. The list of systems the tools were tested on include Norgay at NASA Ames, Mana at MHPCC, Harold at ARL's DSRC, and Garnet at ERDC's DSRC. To simplify testing and maintenance, a build check and a run test were added to Helios. The build check just verifies that the tools built properly with Helios and the proper Python modules are able to be imported. The run test verifies the coprocessing output from a sphere simulation.

Parallel Performance Testing of the CoProcessing Library

DISTRIBUTION STATEMENT A
Distribution Statement A: Distribution is approved for public release; distribution is unlimited.

To make sure that the coprocessing library can be properly used for large parallel runs, we performed tests on the 16 rack IBM BlueGene/L located at the Computational Center for Nanotechnology Innovations at Rensselaer Polytechnic Institute. The runs were done with PHASTA, a multi-purpose CFD code, linked with the ParaView coprocessor via the python adapter. The first series of runs were performed with a 50 million element mesh that was partitioned for executing on 256, 512, 1,024, and 1,800 nodes. The second series of runs used a 133 million element mesh partitioned for 512, 1,024, 2,048, 4,096, 8,192, and 16,384 nodes. Each run consisted of 10 simulation time steps. A time step begins with the main PHASTA computation followed by a call to the adapter. The adapter step converts data stored in Fortran arrays to C arrays suitable for VTK data structures. Next the coprocessor routine is called. For our tests, the coprocessor routine executed the integrate attributes filter, slice filter, and a file writer. The integrate attributes filter and slicer filter require computation time proportional to the number of input mesh cells. The integrate attributes filter also has an additional global communication step. On the BlueGene/L the MPI communication time for the integrate attributes filter was small and the filter displayed near linear scaling as the job size was increased. Due to some uneven cell distribution when increasing job size caused by the position of the slice plane through the data, the slice filter did not show exact linear scaling for all job sizes.
Results for the larger mesh on more nodes seem to follow the trend but a bit more care needs to be taken with performing file I/O on the runs with larger amounts of nodes. The runs on 16,384 nodes failed due to I/O issues. The PHASTA simulator aborted while loading its input data, and this fault occurred even when the coprocessor was removed. We believe the fault is due to the file system stress for that many nodes. The performance of the file writer varied from node to node. The BlueGene/L performs disk I/O only at I/O nodes. Sets of compute nodes are organized into processing sets that share an I/O node. There are typically 32 compute nodes and 1 I/O node in a processing set. Our coprocessing test runs did not make an effort to optimize I/O load with respect to processing sets, instead we executed the data writer on all nodes that contained a non-zero amount of cells. The results for the 50 million and 133 million element mesh runs are in the tables below. The tables show the maximum time, in seconds, for each filter of any single process.

	Integrate Attributes	Slice
256 nodes	1.14	3.13
512 nodes	0.58	1.86
1024 nodes	0.29	0.96
1800 nodes	0.17	0.63

Table 2: 50 Million element mesh run results.

	Integrate Attributes	Slice
512 nodes	1.34	3.01
1,024 nodes	0.67	1.65
2,048 nodes	0.34	0.83
4,096 nodes	0.17	0.52
8,192 nodes	0.09	0.26
16,384 nodes	n/a	n/a

Table 3: 133 Million element mesh run results.

The figure below shows the maximum time for any process to perform a full simulation time step with coprocessing as well as the time dedicated to coprocessing only. This is shown for each time step.

DISTRIBUTION STATEMENT A
Distribution Statement A: Distribution is approved for public release; distribution is unlimited.

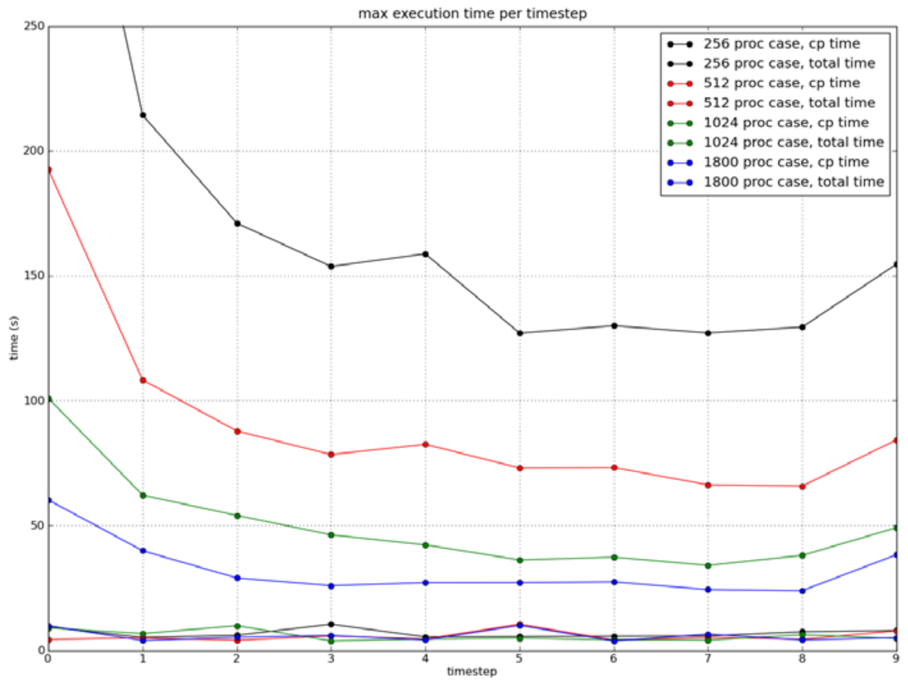


Figure 15: Timing results.

Promoting CoProcessing

So far we have shown the work involved in developing tools for analysis of CFD simulation results as well as the new coprocessing tools in ParaView Version 3.8.1. As ParaView is an open source product, a good way to demonstrate its success is by showing how it is getting used. Through the project we have developed a set of adaptors to convert the grid and field information from simulation data structures to VTK data structures as well as interfacing the simulation code to the coprocessing library (shown in Figure 2). The list of simulation codes that we have worked on adaptors for include PHASTA, HELIOS, Overflow, AVUS, CTH, and AcuSolve. Browsing through the quarterly progress reports will show coprocessing results from many of these codes. Developing adaptors for several simulation codes has helped in ensuring that the coprocessing library can effectively work with a variety of grid types (unstructured, rectilinear, uniform, and AMR grids as well as mixed grid types with overlapping sections using blanking) and languages (Fortran, C, C++, and Python). In addition, collaborators from Sandia National Labs have implemented an adaptor for their own simulation code NPIC. The advantage of the Sandia collaboration is that through using the coprocessing tools, they have had very helpful feedback on improving the software. There has been interest from others on the ParaView users email list as well asking questions about implementing an adaptor for their own simulation codes. This interest has led to writing a wiki page (http://paraview.org/Wiki/CoProcessing) about both using the plugin for creating coprocessing scripts as well as developing adaptors between simulations codes and the coprocessing library. Besides the wiki and mailing lists, we have also done a tutorial/demonstration of using HELIOS with the coprocessing library at NASA Ames. There have been tutorials for using the coprocessing tools at SC10 in NewOrleans and SC11 in Seattle and will be another at SC12 in Salt Lake City.. There was also a paper and a presentation on the coprocessing tools at the Large-Scale Data Analysis and Visualization conference (LDAV 2011). This was a collaboration between Sandia National Laboratories, University of Colorado (Ken Jansen led this effort with his PHASTA simulation code).
To clearly differentiate ParaView's coprocessing library, we have branded it Catalyst. A logo and a combination mark have been created for it and are shown in Figures 16 and 17, respectively. We are in the process of creating a specific web-site for it as well which will be http://catalyst.paraview.org.

DISTRIBUTION STATEMENT A
Distribution Statement A: Distribution is approved for public release; distribution is unlimited.

Figure 16: Catalyst Logo.

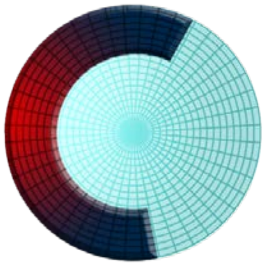


Figure 17: Catalyst Combination Mark.

Conclusions

We have shown that the coprocessing tools developed with the funds from this project can be a useful tool for analyzing CFD simulation results from large runs. We believe that for those users that are comfortable using ParaView and have sufficient knowledge about the physics that they want analyzed, these coprocessing tools can be effectively used to simplify and speed up their work flow. As users get more comfortable with ParaView, they also have the ability to expand the analysis as well as perform small modifications to their python scripts to program more logic into the coprocessing results. For non-programmers these tools can still be used as long as someone with sufficient knowledge about the simulation code data structures and VTK data structures constructs the adaptor code needed to hook the two together. The main advantage for the end user though is that they will be able to run their simulation codes at leading edge computing facilities while still being able to efficiently analyze the results from these massive simulation runs, which was the main goal of the project.

DISTRIBUTION STATEMENT A
Distribution Statement A: Distribution is approved for public release; distribution is unlimited.